DOI https://doi.org/10.15407/csc.2023.03.061
UDC 004.42

V.V. ZOSIMOV, PhD (Eng.) Sciences, Professor, Department of Applied Information Systems, Taras Shevchenko National University of Ukraine, ORCID: https://orcid .org/0000-0003-0824-4168, Bohdan Hawrylyshyn, str. 24, Kyiv, 04116, Ukraine, zosimovvv@gmail.com

O.S. BULGAKOVA, PhD (Ehg.), Sciences, Associate Professor, Department of Applied Information Systems, Taras Shevchenko National University of Ukraine, ORCID: https://orcid .org/0000-0002-6587-8573; Bohdan Hawrylyshyn str. 24, Kyiv, 04116, Ukraine, sashabulgakova2@gmail.com

OPTIMIZING COMPUTATIONAL PERFORMANCE WITH OPENMP PARALLEL PROGRAMMING TECHNIQUES

The article presents a study of parallel computing, specifically comparing the performance of OpenMP in C++ and Python. Furthermore, the technologies of OpenMP and TPL (C++, C#) are contrasted. Performance indicators were established that showcase the advantages and dis-advantages of each methodology. In addition to the numerical data, the research provides insights into the peculiarities of each parallel computing model, which can assist developers in choosing the right tool.

***Keywords**: Parallel computing, OpenMP, TPL, C++, Python, threads, parallelism.*

Introduction

In the modern world, computational technologies play a pivotal role in various spheres of human activity. With the advent of multi-core and multiprocessor systems, there arose a need for efficient programming to fully harness the potential of these systems. However, parallel programming is a complex task that requires specialized skills and tools.

OpenMP (Open Multi-Processing) is a popular specification for parallel programming in C, C++, and Fortran languages [1]. However, other implementations or adaptations of OpenMP have emerged for different languages, such as Python, Java, C#, and others. OpenMP allows developers to seamlessly integrate parallel constructs into their applications, ensuring scalability and portability.

Many researchers actively explore the capabilities and limitations of OpenMP, as well as the development of new methodologies and techniques aimed at performance enhancement. For instance, in [2], the intricacies of parallel programming using TBB are discussed and compared with OpenMP. In [3], authors introduced a framework (SPar) as a solution for simplifying the development of parallel stream processing applications based on OpenMP, demonstrating that this solution is efficient in terms of performance and code complexity reduction. [4] delves into the challenge of optimizing software for heterogeneous supercomputers, especially those with GPU support. The authors suggest using a task-based programming model with MPI+OpenMP (using the target directives from versions 4.0 and 4.5) to achieve a high degree of asynchronous operations and enhance performance on heterogeneous architectures. Article [5] introduces a new approach to managing implicit

parallelism in library functions. Authors propose using "moldable tasks" to express this implicit parallelism and dynamically adjust the parallelization degree of a task. For this, a new construct, task-moldable, is introduced which generates multiple tasks from a single function call.

Based on the above research, the significance of parallel computations in modern high-performance systems becomes evident. Using OpenMP as a tool for controlling parallel computations remains a key aspect in application performance optimization. Against this backdrop, the goal of our article is to implement parallel computations using OpenMP tools, and to conduct comparative evaluation of program execution with varying implementations: with OpenMP, without its use, and in comparison with other distributed computation technologies. Thus, we aim to provide an understanding of the actual benefit of incorporating OpenMP into the software development process.

Features and Application of the OpenMP Parallel Programming Standard

OpenMP (Open Multi-Processing) is a standard for parallel programming in shared-memory systems that allows tasks to be distributed among various processors within a computer or across a network of computers [1]. OpenMP is based on the idea of adding directives to the source code, instructing the compiler on how to distribute the execution of different parts of the program among processors. For example, the directive "#pragma omp parallel" creates a region of parallel code execution that follows it, i.e., the code between the curly braces that follow the directive. The OpenMP compiler automatically distributes this code among the processors available in the system.

For better control over resource distribution, OpenMP provides the ability to specify different types of task schedules among processors. For instance, the directive "#pragma omp for" indicates that the loop should be divided among processors. For enhanced performance, scheduling parameters can be set, such as the number of processors that can participate in the distribution and the number of iterations to be processed by each processor.

OpenMP also supports synchronization between processors using directives like "#pragma omp barrier", which blocks code execution on all processors until all processors reach a given point.

Since OpenMP is a standard supported by many compilers, it allows easy code execution distribution across different shared-memory systems, including various operating systems and processor architectures. Moreover, OpenMP is relatively easy to use and doesn't demand extensive knowledge in parallel programming.

One of the main advantages of OpenMP is the ability to combine parallel programming with conventional sequential programming. This means that sections of the program suitable for parallel execution can be easily isolated and distributed among processors without altering the rest of the program.

However, OpenMP has its limitations. It doesn't support distributed computations, meaning one can't distribute code execution among computers that don't share memory. Additionally, effectively utilizing OpenMP can be a challenging task since it requires considering various factors like the number of available processors, the size of the task, iteration scheduling, and so on.

Overall, OpenMP is a powerful and convenient tool for parallelizing programs on shared-memory systems. Its ease of use and widespread adoption make it popular among developers looking to boost the performance of their programs on multi-core processors.

Methodologies for Evaluating Parallel Computing Efficiency

For the purpose of evaluating the technology's efficiency, we decided to choose a task that is not complicated to implement (so that on various programming languages, the solution can be relatively easily implemented without the use of specific libraries). However, this task should have an increasing time component and the ability to be divided into separate computational processes that can run concurrently. The task condition: develop a software application that calculates the sum of the first N terms of a given sequence.

The study was conducted in stages:

Stage 1. Selection of programming languages. To begin the task, it was necessary to choose the programming languages on which this experiment, described in the assignment, would be conducted. The following languages were selected:

- Low-level language C++ with strict typing and manual memory management;
- High-level language Python3 with dynamic typing and automatic memory management.

Stage 2. Implementation.

Stage 3. Investigation.

For the selected programming languages, the following comparison criteria were determined:

- Distributed computing technology: sequential execution (without parallelism), threads, and OpenMP.

Threads are a powerful tool for improving application performance and responsiveness.

It is the ability of a CPU, or a single core in a multi-core processor, to provide multiple threads of execution concurrently. This can be achieved either by time-slicing (where a single processor switches between different threads) or by truly parallel execution if there are multiple processors/cores available [6]. OpenMP provides a high-level API for parallel programming based on compiler directives, making it relatively easy to use. It's designed for portability, and code written using OpenMP can easily move between different platforms that support this standard. A key feature of OpenMP is its fork-join parallelism model, where the main thread branches off into multiple threads when entering parallel sections and then joins back after their completion. Thread management, their creation, and termination are fully automated in OpenMP. Synchronization is also simplified thanks to built-in directives.

On the other hand, when working with basic threads, such as POSIX threads or Windows Threads, a programmer requires a more detailed and low-level thread management. This includes manually creating, managing the thread's lifecycle, and implementing synchronization mechanisms. Such control might offer more flexibility but also demands a greater attention to details, potentially increasing programming complexity and error likelihood.

In conclusion, the choice between OpenMP and basic threads depends on the specific requirements of the task and the programmer's preferences. OpenMP offers simplicity and portability, while direct thread management provides flexibility at a more granular level.

- number of members (number of iterations);
- execution time of the program.

Results

Table 1 presented the code fragment from the program's execution at stage 2 using threads and OpenMP .

In the C++ (Threads) code snippet, a function called computeSum was created, which takes the starting and ending index and a reference to the variable S. This function calculates the sum of the Fi values for the specified index range.

In the main function (main), we first ask the user to enter the value of n, as before. A vector of threads is then created, and the number of threads to use is determined based on the number of cores available in the system (thread::hardware_concurrency()). The index range for calculation is equally divided among the threads, and each thread is then launched using the computeSum function.

After all the threads have completed their calculations, the program waits for all the threads to finish using the join method. Finally, the algorithm calculates the time spent on the computation and outputs the result to the console.

In the C++ (OpenMP) code snippet, OpenMP is used to parallelize the loop that calculates the sum of the series. The omp parallel for directive is used to instruct OpenMP to parallelize the loop, and the reduction(+:S) code is used to accumulate partial results into the final result. Each thread executes a portion of the loop iterations, with the number of iterations per thread determined by OpenMP automatically.

Therefore, the #pragma omp parallel for directive creates a team of threads and distributes the loop iterations among them. The reduction operation ensures that the sum S is computed in a thread-safe manner, creating a separate copy of S for each thread and accumulating the results of each thread into the final value after the loop completes.

Table 1. **Code fragment using threads and OpenMP**

Code fragment C++	Code fragment Python3
Threads	

...
int main() {
 	
time_t time_1, time_2;
long int n;
long double S = 0;
printf("Enter <n>: ");
scanf("%ld", &n);
time(&time_1);
vector<thread> threads;
 	
int num_threads = thread::hardware_concurrency();
long int chunk_size = n / num_threads;
 	
for (int i = 0; i < num_threads; i++) {
long int start = i * chunk_size;
long int end = (i + 1) * chunk_size;
if (i == num_threads - 1) {
end = n;
}
threads.push_back(thread(computeSum, start, end, ref(S)));
}
 	
for (auto& t : threads) {
t.join();
}
...
	...
	def compute_sum(start, end, result):
	
	local_sum = 0
	for i in range(start, end):
	local_sum += .....// formula of layer
	result.append(local_sum)
	
	n = int(input('Enter <i>: '))
	time1 = time.time()
	S = []
	num_threads = ...
	chunk_size = n // num_threads
	threads = []
	
	for i in range(num_threads):
	start = i * chunk_size
	end = (i + 1) * chunk_size if i < num_threads - 1 else n
	thread = thread-ing.Thread(target=compute_sum, args=(start, end, S))
	threads.append(thread)
	thread.start()
	
	for thread in threads:
	thread.join()
	....

OpenMP	

...
time(&time_1);
#pragma omp parallel for reduction(+:S)
for (int i=0; i<n; i++) {
S += ...// formula;
}
time(&time_2);
printf("Sum of Fi's: %Lf \n", S);
printf("Time: %li s\n", time_2 - time_1);
...
	...
	num_threads = ...// num_threads
	chunk_size = n // num_threads
	results = np.zeros(num_threads)
	
	with
	omp.parallel(num_threads=num_threads):
	thread_num = omp.thread_num()
	start = thread_num * chunk_size
	end = (thread_num + 1) * chunk_size if thread_num < num_threads - 1 else n
	results[thread_num] = com-pute_sum(start, end)
	result = np.sum(results)
	...

In the Python3 (Threads) code snippet, a function named compute_sum was created that takes the starting index, the ending index, and a list for storing the local sum computed by each thread. This function calculates the sum of the Fi values for the specified index range and adds the local sum to the final result.

In the main function, we first ask the user to enter the value of n, as before. Then an empty list S is created to store the local sums computed by each thread. The number of threads is determined and the index range for computation is evenly divided among them.

A list of threads is created, and for each thread, a new thread object is instantiated using the Thread constructor from the threading module. The compute_sum function is passed as the thread's target along with the starting index, ending index, and the S list as arguments.

After all the threads have completed their calculations, the program waits for all the threads to finish using the join method. Finally, the S list is summed up to get the final result, the elapsed time is calculated, and both the result and the elapsed time are displayed.

To integrate with OpenMP in the Python3 code, the pyopenmp dependency was added. Instead of using a list to store results, a numpy array named results was used. This helps to optimize memory access and prevent conflicts when accessing shared resources. In the main code segment, the construction with omp.parallel(num_threads=num_threads): is used to create a parallel region. Within this region, each thread computes its portion of the task, as it gets a unique number using omp.thread_num(), which helps in determining which portion of the range it should process. After computations, each thread stores its partial sum in a separate cell of the results array. After the parallel section ends, all partial sums are combined to get the overall result.

Overall, the rewritten program is much simpler and more concise than the one with regular threads, thanks to the powerful parallelization capabilities of OpenMP.

Table 1 presents the results of Stage 3.

Based on the obtained data, conclusions were made regarding the efficiency of different types of parallel computing programs for a specific task calculating the sum of iterations of a given function in threads. Upon analyzing, it was found that the program written in C++ using OpenMP performs voluminous computations somewhat faster than the program using native threads, and much faster than the program without threads, while obtaining identical, that is, the only correct answers.

While the capabilities of OpenMP are vast and impressive, it would be intriguing to juxtapose it with other parallel computing technologies to have a holistic understanding of the advancements in this domain. One such counterpart that merits consideration is the Task Parallel Library (TPL) in C# [7].

The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces. It simplifies the process of adding parallelism and concurrency to applications. TPL, built on the low-level primitives of the .NET Framework, integrates efficiently with the C# language and supports a variety of parallel operations like parallel loops and parallel tasks. TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. Table 3 shows a comparison of the two technologies.

Based on the presented table 4, it is evident that C++ using OpenMP typically emerges as the fastest processing method compared to the other approaches. This technology demonstrates especially commendable results with larger values of N, highlighting its strong scalability. On the other hand, C# programs utilizing TPL perform faster than equivalent C# programs that use native threads, underscoring the effectiveness of TPL within the context of asynchronous programming in C#. Although, at smaller N values, the difference between C++ with OpenMP and C++ using native threads isn't starkly pronounced, as N increases, the programs with C++ and OpenMP become considerably more efficient. Moreover, C++ programs generally execute more swiftly than their C# counterparts. However, it's noteworthy that the performance gap narrows when TPL is adopted for C#. As N grows, the performance disparity between the different parallelization technologies becomes more pronounced, especially when contrasting C++ (OpenMP) with other methods. In conclusion, the choice among various parallelization technologies will hinge on specific project requirements, but the data from this table emphasizes the advantages of deploying OpenMP in C++ for computation-intensive tasks.

Table2. **Parallel technology comparison**

	N	Time (without parallelism), ms	Time (threads), ms	Time (OpenMP), ms
C++	100	42	6	6
	1000	39	6	5
	10000	52	7	6
	100000	68	13	7
	1000000	75	64	33
	10000000	747	564	242
	100000000	6892	5248	2234
Python3	100	70	25	20
	1000	140	45	35
	10000	700	220	170
	100000	6892	2101	1700
	1000000	71200	19168	16983
	10000000	680103	215000	165000
	100000000	timeout	Timeout	timeout

Table 3. **Comparison between OpenMP and TPL**

Comparison	OpenMP	TPL
Language Specificity	Platform-independent. Designed for C, C++, and Fortran.	Tailored for C# and the .NET framework.
Ease of Use and Integration	Uses compiler directives, making parallelism straightforward in code. However, requires compiler support.	Uses an asynchronous programming model which may require a shift in thinking for developers familiar with traditional threading models.
Performance	The performance largely depends on the compiler and system capabilities. Generally provides good speedup for parallelizable tasks.	Being deeply integrated with .NET, TPL can leverage the platform's optimizations. Might provide better performance for specific .NET tasks.
Flexibility and Granularity	Offers fine-grained control over threading and parallel regions.	Abstracts many complexities of threading, focusing on higher- level task-based parallelism. Might be less flexible in some scenarios compared to OpenMP.

Conclusion

Exploring parallel computing methodologies and their performance across different programming models has provided insightful observations. When juxtaposing OpenMP with native threading in C++ and Task Parallel Library (TPL) in C#, OpenMP in C++ consistently demonstrated superior performance, especially for larger datasets. C#'s TPL also showcased commendable efficiency, surpassing native C# threading and narrowing the performance gap when compared to C++ solutions. While the advantages of using OpenMP in C++ for computation-intensive tasks were clear, it was also evident that language and platform specificity play pivotal roles in determining parallelization performance. Asynchronous programming models,

Table 4. **Parallel technology comparison: OpenMP and TPL**

N	Time C#(TPL), ms	Time C++ (OpenMP), ms	Time C++ (threads), ms	Time C# (threads), ms
100	10	6	6	42
1000	7	5	6	39
10000	8	6	7	52
100000	18	7	13	68
1000000	79	33	64	75
10000000	684	242	564	747
100000000	4427	2234	5248	6892

such as TPL, offer significant boosts within their specific contexts, like C#. In essence, while C++ with OpenMP stood out as an optimal solution for parallel processing in this study, the choice of technology is influenced by the broader ecosystem and specific requirements of each project.

Going forward, broadening the spectrum of technologies investigated would be beneficial. Incorporating frameworks like MPI or CUDA could provide deeper insights into the parallel processing landscapes. Further, considering the influence of underlying hardware components on performance would be essential. Different CPU architectures, memory management strategies, and even the role of GPUs in parallel computation could be vital areas of study. Finally, real-world application testing, taking into account the complexities of actual software development scenarios, would offer a holistic perspective on the applicability and performance of these parallel technologies.

REFERENCES

1. OpenMP, [online]. Available at: https://www.openmp.org/ [Accessed: 02 Aug. 2023].
2. *Reinders, J.*, 2007. "Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism".
3. *Hoffmann, R.B., Lff, J., Griebler, D.* et al. "OpenMP as runtime for providing high-level stream parallelism on multi-cores". JSupercomput 78, 7655–7676, 2022. https://doi.org/10.1007/s11227-021-04182-9.
4. *Ferat, M., Pereira, R., Roussel, A., Carribault, P., Steffenel, LA., Gautier, T.* "Enhancing MPI+OpenMP Task Based Applications for Heterogeneous Architectures with GPU Support" OpenMP in a Modern World: From Multi-device Support to Meta Programming. IWOMP 2022. Lecture Notes in Computer Science, vol 1352. Springer, Cham. 2022. https://doi.org/10.1007/978-3-031-15922-0_1.
5. *Polet, P., Fantar, R., Gautier, T.* "Introducing Moldable Tasks in OpenMP" Advanced Task-Based, De-vice and Compiler Programming. IWOMP 2023. Lecture Notes in Computer Science, vol 14114. Springer, Cham, 2023. https://doi.org/10.1007/978-3-031-40744-4_4.
6. Parallel Programming Using Threads, [online]. Available at: https://www.oreilly.com/library/view/parallel-and-concurrent [Accessed 20 Aug. 2023].
7. Task Parallel Library (TPL), [online]. Available at: https://learn.microsoft.com [Accessed: 21 Aug. 2023].

Received 19.09.2023

В.В. Зосімов, д.т.н., професор, Київський національний університет імені Тараса Шевченка,
Україна, ORCID: https://orcid .org/0000-0003-0824-4168, Scopus Author ID 57188682230,
04116, м. Київ, вул. Богдана Гаврилишина, 24, Україна,
zosimovvv@gmail.com
О.С. Булгакова, к.т.н., доцент, Київський національний університет імені Тараса Шевченка,
Україна, ORCID: https://orcid .org/0000-0002-6587-8573; Scopus Author ID 57188687900,
04116, м. Київ, вул. Богдана Гаврилишина, 24, Україна,
sashabulgakova2@gmail.com

ОПТИМІЗАЦІЯ ОБЧИСЛЮВАЛЬНОЇ ПРОДУКТИВНОСТІ ЗА ДОПОМОГОЮ МЕТОДІВ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ *OPENMP*

Вступ. Поява багатоядерних та багатопроцесорних систем вимагає ефективних програмних підходів для розкриття їх повного потенціалу. Однак паралельне програмування залишається складним завданням, що вимагає специфічної експертизи та інструментів. *OpenMP*, відомий стандарт для паралельного програмування, пропонує розробникам можливість легко впроваджувати паралельні конструкції в їхні додатки, гарантуючи масштабованість та адаптивність. Багато досліджень зосереджено на потенціалі та обмеженнях *OpenMP*, пропонуючи нові методи для підвищення продуктивності. Усвідомлюючи критичну роль паралельних обчислень у сучасних високопродуктивних системах, ця стаття має на меті продемонструвати впровадження паралельних обчислень із використанням *OpenMP*. Ми маємо намір порівняти виконання програми в різних сценаріях: із використанням *OpenMP*, без нього та у порівнянні з іншими методами паралельних обчислень. У такий спосіб, ми прагнемо висвітлити реальні переваги інтеграції *OpenMP* у процес розробки програмного забезпечення.

Мета статті. Дослідження можливостей паралельних обчислень за допомогою *OpenMP*, порівняння його продуктивності в різних сценаріях використання, а також виявлення переваг та особливостей інтеграції *OpenMP* у процесі розробки програмного забезпечення.

Методи. Системний підхід, аналітичний метод, експериментальний метод.

Результати. При порівнянні *OpenMP* з рідним потокуванням в *C++* та *Task Parallel Library* (*TPL*) в *C#*, *OpenMP* в *C++* постійно продемонстрував відмінну продуктивність, особливо для більших наборів даних. *TPL* в *C#* також виявив заслужену ефективність, перевершивши рідне потокування C# та звузивши відставання у продуктивності порівняно з рішеннями на *C++*. Хоча переваги використання *OpenMP* в *C++* для завдань, що потребують інтенсивних обчислень, були очевидними, стало ясно, що мовна та платформенна специфіка відіграють ключові ролі у визначенні продуктивності паралелізації. Асинхронні моделі програмування, такі як *TPL*, пропонують значуще прискорення у своїх конкретних контекстах, наприклад, у *C#*. Здебільшого, хоча *C++* з *OpenMP* виділявся як оптимальне рішення для паралельної обробки в цьому дослідженні, вибір технології залежить від ширшого екосистемного контексту та специфічних вимог кожного проєкту.

Висновки. Представлено дослідження паралельних обчислень, зокрема порівняння продуктивності *OpenMP* у *C++* і *Python*. Крім того, протиставляються технології *OpenMP* і *TPL* (*C++*, *C#*). Було встановлено показники ефективності, які демонструють переваги та недоліки кожної методології. Окрім числових даних, дослідження дає змогу зрозуміти особливості кожної моделі паралельних обчислень, що може допомогти розробникам у виборі правильного інструменту.

Ключові слова: *паралельні обчислення, OpenMP, TPL, C++, Python, потоки, паралелізм.*